CONVENTIONAL SEMANTIC APPROACHES

Grigore Rosu

CS422 – Programming Language Design

Conventional Semantic Approaches

A language designer should understand the existing design approaches, techniques and tools, to know what is possible and how, or to come up with better ones. This part of the course will cover the major PL semantic approaches, such as:

- Big-step structural operational semantics (Big-step SOS)
- Small-step structural operational semantics (Small-step SOS)
- Modular structural operational semantics (Modular SOS)
- Reduction semantics with evaluation contexts
- The chemical abstract machine
- Denotational semantics

IMP – A Simple Imperative Language

We will exemplify the conventional semantic approaches by means of IMP, a very simple non-procedural imperative language, with

- Arithmetic expressions
- Boolean expressions
- Conditional statements
- While loop statements

IMP Syntax

Int	::=	the domain of (unbounded) integer numbers, with usual operations on them
Bool	::=	the domain of Booleans
Id	::=	standard identifiers
AExp	::=	*Int*
	|	*Id*
	|	*AExp* + *AExp*
	|	*AExp* / *AExp*
BExp	::=	*Bool*
	|	*AExp* <= *AExp*
	|	not *BExp*
	|	*BExp* and *BExp*
Stmt	::=	skip
	|	*Id* := *AExp*
	|	*Stmt* ; *Stmt*
	|	if *BExp* then *Stmt* else *Stmt*
	|	while *BExp* do *Stmt*
Pgm	::=	vars **List**{*Id*} ; *Stmt*

Suppose that, for demonstration purposes, we want "+" and "/" to be nondeterministically strict, "<=" to be sequentially strict, and "and" to be short-circuited.

IMP Syntax in Maude

- See files in **`imp-syntax.zip`** for remaining details

```
mod IMP-SYNTAX is including PL-INT + PL-BOOL + PL-ID .
--- AExp
  sort AExp .  subsorts Int Id < AExp .
  op _+_ : AExp AExp -> AExp [prec 33 gather (E e) format (d b o d)] .
  op _/_ : AExp AExp -> AExp [prec 31 gather (E e) format (d b o d)] .
--- BExp
  sort BExp .  subsort Bool < BExp .
  op _<=_ : AExp AExp -> BExp  [prec 37 format (d b o d)] .
  op not_ : BExp -> BExp [prec 53 format (b o d)] .
  op _and_ : BExp BExp -> BExp [prec 55 format (d b o d)] .
--- Stmt
  sort Stmt .
  op skip : -> Stmt [format (b o)] .
  op _:=_ : Id AExp -> Stmt [prec 40 format (d b o d)] .
  op _;_ : Stmt Stmt -> Stmt [prec 60 gather (e E) format (d b noi d)] .
  op if_then_else_ : BExp Stmt Stmt -> Stmt [prec 59 format (b o bni n++i bn--i n++i --)] .
  op while_do_ : BExp Stmt -> Stmt [prec 59 format (b o d n++i --)] .
--- Pgm
  sort Pgm .
  op vars_;_ : List{Id} Stmt -> Pgm [prec 70 format (nb o d ni d)] .
endm
```

IMP State

- Most semantics need some notion of *state*
- A state holds all the semantic ingredients to fully define the meaning of a given program or fragment of program
- For IMP, a state is simply a *partial finite-domain function* from identifiers to integer values, written

$$\sigma : Id \rightharpoonup Int$$

- We write the domain of such functions, say *State*, as

$$[Id \rightharpoonup Int]^{finite}$$

or

$$\mathbf{Map}\{Id \mapsto Int\}$$

Lookup, Update and Initialization

- We may write states by enumerating each identifier binding. For example, the following state binds x to 8 and y to 0:

$$\sigma \;=\; x \mapsto 8, y \mapsto 0$$

- Typical state operations are lookup, update and initialization
- *Lookup*

$$\_(\_) : State \times Id \to Int$$

- *Update*

$$\_[\_/\_] : State \times Int \times Id \to State$$

- *Initialization*

$$\_ \mapsto \_ : \mathbf{List}\{Id\} \times Int \to State$$

IMP State in Maude

- See file **`state.zip`**

```
mod STATE is including PL-INT + PL-ID .
  sort State .
  op _|->_ : List{Id} Int -> State [prec 0] .
  op .State : -> State .
  op _&_ : State State -> State [assoc comm id: .State format(d s s d)] .
  op _(_) : State Id -> [Int] [prec 0] .        --- lookup
  op _[_/_] : State Int Id -> State [prec 0] .  --- update

  var Sigma : State .  var X X' : Id .  var Xl : List{Id} .  var I I' : Int .

  eq (Sigma & X |-> I)(X) = I .
  eq (Sigma &  X |-> I)[I' / X ] = (Sigma & X |-> I') .
  eq (X,X',Xl) |-> I = X |-> I & X' |-> I & Xl |-> I .
  eq .List{Id} |-> I = .State .
endm
```